5

Arasöz: API İşlemi

Not: Arasözler

Arasözler, belirli bir işletim sistemi API'lerine ve bunların nasıl kullanılacağına odaklanılması da dahil olmak üzere sistemlerin daha pratik yönlerini kapsayacaktır. Pratik şeyleri sevmezseniz, bu araya girmeyi atlayabilirsiniz. Ancak pratik şeyleri sevmeniz gerekir, çünkü bunlar genellikle gerçek hayatta faydalıdır; örneğin şirketler, pratik olmayan becerileriniz için sizi genellikle işe almazlar.

Bu aralarda UNIX sistemlerinde süreç oluşturma konusunu ele alıyoruz. UNIX, bir çift sistem çağrıyla yeni bir süreç oluşturmanın en ilginç yollarından birini sunar: Fork () ve exec (). Üçüncü bir rutin olan Wait (), oluşturulduğu işlemin tamamlanmasını beklemek isteyen bir süreç tarafından kullanılabilir. Şimdi bu arayüzleri daha ayrıntılı olarak sunuyoruz ve bizi motive etmek için birkaç basit örnek sunuyoruz. Bu nedenle, sorunumuz:

PÜF NOKTASI : SÜREÇ OLUŞTURMA VE KONTROL ETME

İşletim sistemi süreç oluşturma ve kontrol için hangi arayüzleri sunmalıdır? Bu arabirimler güçlü bir işlevi, kullanım kolaylığını ve yüksek performansı sağlayacak şekilde nasıl tasarlanır?

5.1 `fork()` Sistem Çağrısı

`fork()` sistem çağrısı, yeni bir süreç oluşturmak için kullanılır [C63]. Ancak, önceden uyarılın: Bu, kesinlikle şimdiye kadar arayabileceğin en garip rutindir[1]. Daha açık bir şekilde, kodu Şekil 5.1'de gördüğünüz gibi görünen bir koşu programına sahip olmalısın; kodu inceleyin veya daha iyisi yazın ve kendi başınıza çalıştırın!

[1]Pekala, bunu kesinlikle bilmediğimizi itiraf ediyoruz; kimse aramadığında hangi rutinleri aradığınızı kim biliyor? Ancak çatal (), rutinleriniz ne kadar olağandışı olursa olsun oldukça garip.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main(int argc, char *argv[]) {
6    printf("hello world (pid:%d)\n", (int) getpid());
7    int rc = fork();
8    if (rc < 0) {
9      // fork failed
10     fprintf(stderr, "fork failed\n");
11     exit(1);
12   } else if (rc == 0) {
13     // child (new process)
14     printf("hello, I am child (pid:%d)\n", (int) getpid());
15   } else {
16     // parent goes down this path (main)
17     printf("hello, I am parent of %d (pid:%d)\n",
18             rc, (int) getpid());
19   }
20   return 0;
21 }
22
```

Şekil 5.1: **`fork()` çağrısı (`p1.c`)**

Bu programı (p1.c olarak adlandırılır) çalıştırdığınızda, aşağıdakileri göreceksiniz:

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

P1.c'de neler olduğunu daha ayrıntılı olarak anlatalım İlk kez çalışmaya başladığında, süreç bir merhaba dünya mesajı yazdırır; bu mesaja dahil olan **süreç tanımlayıcısı (process identifier)** da **PID (PID)** olarak da bilinir. İşlemin PID'si 29146'dir; UNIX sistemlerinde PID, işlemin çalışmasını durdurmak gibi bir işlem yapmak istiyorsa işlemi adlandırmak için kullanılır. Şu ana kadar çok iyi.

Şimdi ilginç bölüm başlıyor. Bu işlem, işletim sisteminin yeni bir süreç oluşturmak için sağladığı fork() sistem çağrılarını çağırır. Tek parça: Oluşturulan işlem, arama işleminin (neredeyse) tam bir kopyasıdır. Yani işletim sistemi için artık p1 programının iki kopyası çalışıyor ve her ikisi de fork() sistem aramasından geri dönmek üzere gibi görünüyor. Yeni oluşturulan süreç (**alt öğe (child)** olarak adlandırılır, **üst öğe (parent)** oluşturulmasına karşılık), bekleyebileceğiniz gibi (dikkat, "merhaba, dünya" mesajı yalnızca bir kez basıldı) main()'da çalışmaya başlamaz; bunun yerine, fork() olarak adlandırılmış gibi hayata geçer.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main(int argc, char *argv[]) {
7    printf("hello world (pid:%d)\n", (int) getpid());
8    int rc = fork();
9    if (rc < 0) {          // fork failed; exit
10     fprintf(stderr, "fork failed\n");
11     exit(1);
12   } else if (rc == 0) { // child (new process)
13     printf("hello, I am child (pid:%d)\n", (int) getpid());
14   } else {               // parent goes down this path (main)
15     int rc_wait = wait(NULL);
16     printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
17            rc, rc_wait, (int) getpid());
18   }
19   return 0;
20 }
21
```

Şekil 5.2: **`fork()` çağrısı ve `wait()` (`p2.c`)**

Şunu fark etmiş olabilirsiniz: Çocuk tam bir kopya değildir. Özellikle, adres alanının kendi kopyasına (yani kendi özel belleğine), kendi kayıtlarına, kendi bilgisayarına ve benzeri bir şeye sahip olsa da, fork () arayana geri döndüğü değer farklıdır. Özellikle, üst öğe yeni oluşturulan alt öğenin PID'sini alırken, alt öğe sıfır olarak bir iade kodu alır. Bu farklılaştırma faydalıdır, çünkü iki farklı durumu ele alan kodu yazmak basittir (yukarıdaki gibi).

Şunu da fark etmiş olabilirsiniz: Çıkış (p1.c) **belirleyici (deterministic)** değildir. Alt işlem oluşturulduğunda, sistemde önemsediğimiz iki etkin işlem vardır: Üst öğe ve alt öğe. Tek CPU'lu bir sistemde çalıştığımızı varsayarak (basitlik için), alt veya üst öğe bu noktada çalışır. Örneğimizde (yukarıdaki), ebeveyn önce mesajını yazdırdı ve çıktı. Diğer durumlarda, bu çıktı izinde gösterdiğimiz gibi tersi de olabilir:

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

En kısa zamanda ayrıntılı olarak ele alacağımız bir konu olan CPU **programlayıcısı (scheduler)**, belirli bir anda hangi işlemin çalışacağını belirler; programlayıcı karmaşık olduğu için genellikle ne yapmayı seçeceğimiz ve

dolayısıyla hangi işlemin önce çalışacağı konusunda güçlü varsayımlar yapamayız. Bu **belirleyici olmama (non- determinism)** durumu, özellikle **çok kanallı programlarda (multi-threaded programs)** bazı ilginç sorunlara yol açar; dolayısıyla kitabın ikinci bölümünde **eş zamanlı (concurrency)** olarak çalıştığımızda çok daha fazla belirleyici olmama durumu görüyoruz.

5.2 `wait()` Sistem Çağrısı

Şimdiye kadar çok şey yapmadık: Yeni bir mesaj basan ve çıkan bir alt işlem yarattık. Bazen, ortaya çıktıkça, bir üst işlemin bir alt işlem sürecinin ne yaptığını bitirmesini beklemek oldukça yararlı olabilir. Bu görev, wait() sistem çağrısı (veya daha eksiksiz alt waitpid()) ile gerçekleştirilir; ayrıntılar için bkz. Şekil 5.2.

Bu örnekte (p2.c), üst işlem, alt yürütmeyi tamamlayana kadar yürütülmesini ertelemek için wait() 'yi arar. Alt işlem bittiğinde, wait() üst işleme döner.

Yukarıdaki koda wait() çağrısı eklemek, çıktının caydırıcı olmasını sağlar. Nedenini görebiliyor musunuz? Devam edin, düşünün.

(düşünmenizi bekliyor..... ve bitti)

Şimdi biraz düşündünüz, sonuç şu:

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (rc_wait:29267) (pid:29266)
prompt>
```

Bu kodla artık alt öğenin her zaman önce baskı yapacağını biliyoruz. Bunu neden biliyoruz? Daha önce olduğu gibi ilk önce de çalıştırılabileceği için, üst öğenin önünde de yazdırılabilir. Ancak, önce ebeveyn çalıştırılmaya başlarsa, hemen `wait()` çağrısı yapar; bu sistem çağrısı, alt öğe çalıştırılıp çıkıncaya kadar geri dönmez[2]. Bu nedenle, ana öğe ilk çalıştırdığında bile, alt öğenin koşmayı bitirmesini bekleyip `wait()` geri döner ve ardından üst öğe mesajını yazdırır.

5.3 Son olarak, `exec()` Sistem Çağrısı

Süreç oluşturma API'sinin son ve önemli bir kısmı `exec()` sistem çağrıdır[3]. Bu sistem çağrısı, arama programından farklı bir programı çalıştırmak istediğinizde yararlıdır. Örneğin, arama `fork()`

[2] Alt öğe çıkmadan önce `wait()` geri dönen birkaç durum vardır; daha fazla ayrıntı için her zaman olduğu gibi MAN sayfasını okuyun. Ve bu kitabın "her zaman önce alt öğe basacak" veya "UNIX, dondurmadan bile dünyanın en iyi yanıdır" gibi yaptığı mutlak ve niteliksiz ifadelere dikkat edin.

[3] Linux'ta, altı adet exec() varyantı vardır: `execl`, `execlp()`, `execle()`, `execv()`, `execvp()`, ve `execvpe()`.Daha fazla bilgi edinmek için adam sayfalarını okuyun.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <sys/wait.h>
6
7  int main(int argc, char *argv[]) {
8    printf("hello world (pid:%d)\n", (int) getpid());
9    int rc = fork();
10   if (rc < 0) {         // fork failed; exit
11     fprintf(stderr, "fork failed\n");
12     exit(1);
13   } else if (rc == 0) { // child (new process)
14     printf("hello, I am child (pid:%d)\n", (int) getpid());
15     char *myargs[3];
16     myargs[0] = strdup("wc");   // program: "wc" (word count)
17     myargs[1] = strdup("p3.c"); // argument: file to count
18     myargs[2] = NULL;           // marks end of array
19     execvp(myargs[0], myargs);  // runs word count
20     printf("this shouldn't print out");
21   } else {              // parent goes down this path (main)
22     int rc_wait = wait(NULL);
23     printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
24            rc, rc_wait, (int) getpid());
25   }
26   return 0;
27 }
28
```

Şekil 5.3: **`fork()` çağrısı, `wait()`, ve `exec()` (`p3.c`)**

p2.c'de yalnızca aynı programın çalışan kopyalarını tutmak istiyorsanız yararlıdır. Ancak, genellikle farklı bir program çalıştırmak istiyorsunuz; /exec () tam olarak bunu yapıyor (Şekil 5.3).

Bu örnekte, alt işlem süreci sözcük sayma programı olan wc programını çalıştırmak için `execvp()` arar. Aslında, p3.c kaynak dosyasında wc çalıştırıyor, böylece dosyada kaç satır, sözcük ve bayt bulunduğunu bize söylüyor:

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
      29     107    1030 p3.c
hello, I am parent of 29384 (rc_wait:29384) (pid:29383)
prompt>
```

`fork()` sistem çağrısı garip; suç ortağı, `exec()`, de bu kadar normal değil. Ne işe yarıyor: Bir yürütülebilir dosyanın adı (örn. wc) ve bazı bağımsız değişkenler (örn. p3.c) göz önünde bulundurulduğunda, bundan kod (ve statik veri) yükler

İpucu: Doğru olanı yapmak (LAMPSON Yasası)
Lampson, saygın "Bilgisayar Sistemleri tasarımı için İpuçları" [L83], "**doğru olanı yapın (Get it right)**. Ne soyutlanma ne de basitlik, bu işi doğru yapmak için bir alternatif." Bazen doğru olanı yapmak zorundasınız ve bunu yaparken, alternatiflerden çok daha iyi bir seçenek. Proses oluşturma için API tasarlamanın birçok yolu vardır; ancak `fork()` ve `exec()` kombinasyonu basit ve son derece güçlüdür. UNIX tasarımcıları işte bu işi doğru yapmıştı. Ve Lampson'un bu kadar sık "doğru anladığı" için, yasaya onur adını vereceğiz.

yürütülebilir ve mevcut kod segmentinin (ve geçerli statik verilerin) üzerine yazar; programın bellek alanının öbek ve yığını ve diğer bölümleri yeniden başlatılır. Ardından, işletim sistemi bu programı çalıştırır ve bu işlemin argümanı olarak tüm argümanlardan geçer. Bu nedenle, yeni bir süreç yaratmaz; daha çok, çalışmakta olan programı (eski adıyla p3) farklı bir çalışma programına (wc) dönüştürür. `exec()` alt öğe, neredeyse p3.c hiç çalışmamış gibi; `exec()` hiçbir zaman geri dönmemiş gibi.

5.4 Neden? API'yi Motive Etme

Elbette, sorabileceğiniz bir soru var: Yeni bir süreç oluşturmanın basit eylemi ne olacak diye neden bu kadar garip bir arayüz oluştururuz? Sonuç olarak, UNIX kabuğu oluştururken `fork()` ve exec() ayrımı çok önemlidir, çünkü `fork()`'a çağrı yapmadan önce `fork()`'a telefon ettikten sonra kabuğun kod çalıştırmasını sağlar; bu kod, çalıştırılacak program ortamını değiştirebilir ve böylece çeşitli ilginç özelliklerin kolayca oluşturulmasına olanak tanır.

Kabuk sadece bir kullanıcı programıdır[4]. Size bir bilgi **istemi (prompt)** gösterir ve ardından bir şey yazmanızı bekler. Daha sonra bir komutu (örn. Yürütülebilir bir programın adı ve tüm bağımsız değişkenler) bu komuta yazarsınız; çoğu durumda, kabuk dosya sisteminde yürütülebilir dosyanın nerede olduğunu gösterir, komutu çalıştırmak için yeni bir alt işlem oluşturmak üzere `fork()` arar, komutu çalıştırmak için `exec()` varyantını arar ve ardından `wait()`'i arayarak komutun tamamlanmasını bekler. Alt öğe işlemi tamamladığında, kabuk `wait()`'den geri döner ve bir sonraki komutunuz için hazır olan bir istemi yeniden yazdırır.

`fork()` ve `exec()` ayrımı, kabuğun çok daha kolay bir şekilde bir sürü faydalı iş yapmasını sağlar. Örneğin:

```
prompt> wc p3.c > newfile.txt
```

[4]Ve birçok kabuk var; tcsh, bash ve zsh bunlardan birkaçı. Birini seçip adam sayfalarını okumanız ve bu konuda daha fazla bilgi edinmeniz gerekir; tüm UNIX uzmanları bunu yapar.

Yukarıdaki örnekte, wc programının çıktısı newfile.txt çıktı dosyasına **yönlendirilir (redirected)** (daha büyük bir işaret, yeniden yönlendirmeyi nasıl ifade ettiğini gösterir). Kabuğun bu görevi yapma şekli oldukça basittir: Alt öğe oluşturulduğunda `exec()` çağırmadan önce kabuk **standart çıktıyı** (**standard output)** kapatır ve `newfile.txt` dosyasını açar. Bu sayede, yakında çalıştırılacak olan wc programından herhangi bir çıktı ekran yerine dosyaya gönderilir.

Şekil 5.4 (sayfa 8) tam olarak bunu yapan bir programı gösterir. Bu yönlendirmenin çalışma nedeni, işletim sisteminin dosya tanımlayıcılarını nasıl yönettiği konusunda varsayımdır. Özellikle UNIX sistemleri, ücretsiz dosya tanımlayıcılarını aramaya başlar. Bu durumda, STDOUT FILENO mevcut ilk FILENO olacaktır ve böylece `open()` çağrıldığında atanır. Alt öğe işlemi tarafından standart çıktı dosyası tanımlayıcısına, örneğin `printf()` gibi rutinler tarafından sonradan yazma işlemi, şeffaf bir şekilde ekran yerine yeni açılan dosyaya yönlendirilir.

p4.c programını çalıştırma çıktısı aşağıda verilmiştir:

```
prompt> ./p4
prompt> cat p4.output
      32     109     846 p4.c
prompt>
```

Bu çıktı hakkında (en az) iki ilginç tidbit göreceksiniz. öncelikle, p4 çalışırken hiçbir şey olmamış gibi görünür; kabuk yalnızca komut istemini yazdırır ve bir sonraki komutunuz için hemen hazırdır. Ancak, böyle bir durum söz konusu değil; p4 programı yeni bir çocuk yaratmak için `fork()` 'ı aradı ve ardından `execvp()` için wc programını çalıştırdı. p4.output dosyasına yönlendirildiğinden ekrana yazdırılmış çıktı göremezsiniz. İkincisi, çıktı dosyasını kesiyorsak, wc'in çalışması için beklenen tüm çıktıların bulunduğunu görebilirsiniz. Harika, değil mi? UNIX boruları da benzer şekilde uygulanır, ancak `pipe()` sistem çağrısı ile. Bu durumda, bir işlemin çıktısı bir kernelli **veriyoluna (pipe)** (örn. Sıraya) bağlanır ve başka bir işlemin girişi aynı boruya bağlanır; bu nedenle, bir işlemin çıktısı sorunsuz bir şekilde bir sonraki komut zincirine girdi olarak kullanılır ve uzun ve kullanışlı komut zincirlerinin bir arada takılması sağlanır. Basit bir örnek olarak, bir dosyada bir sözcük aramayı ve ardından kaç kez söz ettiğini saymayı düşünün; borular ve yardımcı programlar, yağ ve wc ile kolay; tek yapman gereken `grep -o foo file | wc -l` komut istemine yazın ve sonucu çakın.

Son olarak, Process API'yi yüksek bir düzeyde çizerken, öğrenilecek ve sindirilecek bu çağrılarla ilgili çok daha fazla ayrıntı var; örneğin, kitabın üçüncü bölümünde dosya sistemleri hakkında konuştuğumuzda dosya tanımlayıcıları hakkında daha fazla bilgi edineceğiz. Şimdilik, `fork()/exec()` bileşiminin süreçleri oluşturmak ve manipüle etmek için güçlü bir yol olduğunu söylemek yeterli.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

    // now exec "wc"...
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: wc (word count)
    myargs[1] = strdup("p4.c"); // arg: file to count
    myargs[2] = NULL;           // mark end of array
    execvp(myargs[0], myargs);  // runs word count
  } else {
    // parent goes down this path (main)
    int rc_wait = wait(NULL);
  }
  return 0;
}
```

Şekil 5.4: **Yeniden Yönlendirme ile Yukarıdakilerin tümü (p4.c)**

5.5 Süreç Kontrolü ve Kullanıcılar

`fork()`, `exec()`, and `wait()` dışında, UNIX sistemlerinde süreçlerle etkileşim için birçok başka karşılıklı yüz vardır. Örneğin, `kill()` sistem çağrısı, duraklatma, ölme ve diğer faydalı zorunluluklar da dahil olmak üzere bir sürece **sinyal** (**signals**) göndermek için kullanılır. Kolaylık için, çoğu UNIX kabukunda, belirli tuş vuruşu kombinasyonları o anda çalışmakta olan işleme belirli bir sinyal verecek şekilde yapılandırılır; Örneğin, control-c sends a `SIGINT` (kesme) gönderir (normalde bunu sonlandırır) ve Control-z, işlemin orta yürütme sırasında duraklatılmasına neden olan bir `SIGTSTP` (durdurma) sinyali gönderir (bunu daha sonra bir komutla devam ettirebilirsiniz, örneğin birçok kabukta bulunan `fg` yerleşik komutu).

Tüm sinyaller alt sistemi, süreçlere dış olaylar sunmak için, bu sinyalleri ayrı ayrı süreçlerde alma ve işleme yolları ve ayrı ayrı süreçlere ve tüm **süreç gruplarına (process groups)** sinyal gönderme yolları dahil olmak üzere zengin bir altyapı sağlar. Bu iletişim biçimini kullanmak için ,

Not: RTFM — MAN SAYFALARINI OKUYUN

Bu kitapta çoğu kez, belirli bir sistem çağrısı veya kitaplık aramasına atıfta bulunduğumuzda, **kılavuz sayfalarını (manual pages)** veya **insan sayfalarını (man pages)** kısa süreliğine okumanızı söyleriz. MAN sayfaları, UNIX sistemlerinde bulunan orijinal belge biçimidir; **Web (the web)** adı verilen şey mevcut olmadan önce oluşturulduğunu fark edin.

İnsan sayfalarını okurken biraz zaman harcamak, sistem programcısının büyümesinde önemli bir adımdır; bu sayfalarda gizlenmiş tonlarca faydalı zaman bitleri vardır. Özellikle okunması gereken bazı faydalı sayfalar, hangi kabuğun (örneğin **tcsh** veya **bash**) kullanıldığı ve programınızın yaptığı tüm sistem çağrıları için (hangi iade değerlerinin ve hata koşullarının mevcut olduğunu görmek için) adam sayfalarıdır.

Son olarak, insan sayfalarını okumak sizi utanç verici bir şekilde kurtarabilir. İş arkadaşlarınıza `fork()` bazı entrikalarından bahsettiğinizde, "RTFM" yanıtını vermeleri yeterlidir. Bu, iş arkadaşlarınızın sizi Man sayfalarını okumaya teşvik etmenin bir yoludur. RTFM'deki F cümleye biraz renk katar...

bir işlem, çeşitli sinyalleri "yakalamak" için `signal()` sistemi çağrmasını kullanmalıdır; bu şekilde, belirli bir sinyal bir işleme verildiğinde normal çalışmasını askıya alır ve sinyale yanıt olarak belirli bir kod parçası çalıştırabilir. Sinyaller ve çok sayıda entrikaz hakkında daha fazla bilgi edinmek için başka bir yeri okuyun [SR05].

Bu da doğal olarak bir soruya yol açıyor: Kim bir sürece sinyal gönderebiliyor ve kim bunu yapamıyor? Genellikle kullandığımız sistemlerde aynı anda birden fazla kişi kullanılabilir; bu kişilerden biri `SIGINT` (bir süreci yarıda kesmek, muhtemelen sonlandırmak) gibi sinyalleri isteğe bağlı olarak gönderebilirse sistemin kullanılabilirliği ve güvenliği tehlikeye girer. Sonuç olarak, modern sistemler bir **kullanıcı (user)** kavramını güçlü bir şekilde kavramaktadır. Kullanıcı, kimlik bilgilerini oluşturmak için bir parola girdikten sonra, sistem kaynaklarına erişmek için oturum açar. Kullanıcı daha sonra bir veya daha fazla işlem başlatabilir ve bu işlemler üzerinde tam kontrol uygulayabilir (bunları duraklatabilir, öldürebilir vb.). Kullanıcılar genellikle yalnızca kendi süreçlerini kontrol edebilir; genel sistem hedeflerini karşılamak için her kullanıcıya (ve süreçlerine) kaynakları (CPU, bellek ve disk gibi) paketlemesi işletim sisteminin görevidir.

5.6 Kullanışlı Araçlar

Çok kullanışlı birçok komut satırı aracı da vardır. Örneğin, `ps` komutunu kullanarak hangi işlemlerin çalıştığını görebilirsiniz; bazı faydalı bayrakların ps'ye geçmesi için **man sayfalarını (man pages)** okuyun. Aracın üst kısmı da oldukça yardımcı, çünkü sistemin işlemlerini ve ne kadar CPU ve diğer kaynakları görüntülüyor. Komik bir şekilde, birçok kez çalıştırdığınızda en önemli iddia en iyi kaynak ocaktır; belki de bu biraz bir egomanyak. Biraz daha kullanıcı dostu `killall` rastgele göndermek için kullanılabileceği gibi,

Not: Süper Kullanıcı (Kök)

Bir sistemin genellikle sistemi **yönetebilen (administer)** bir kullanıcıya ihtiyacı vardır ve çoğu kullanıcının olduğu gibi sınırlı değildir. Bu işlem kullanıcı tarafından başlatılmamış olsa bile, böyle bir kullanıcı rastgele bir işlemi (örneğin, sistemi bir şekilde kötüye kullanıyorsa) öldürebilir. Bu tür bir kullanıcı, kapatma gibi güçlü komutları da çalıştırabilmelidir (bu, benzersiz bir şekilde sistemi kapatır). UNIX tabanlı sistemlerde, bu özel yetenekler **tam yetkili kullanıcıya (superuser)** (bazen **kök (root)** olarak da adlandırılır) verilir. Çoğu kullanıcı diğer kullanıcı işlemlerini öldüremezken, tam yetkili kullanıcı bunu yapabilir. Kök olmak Spider-Man® olmak gibidir: Büyük güçle büyük sorumluluk [QI15] gelir. Bu nedenle, **güvenliği (security)** artırmak (ve maliyetli hatalardan kaçınmak) için genellikle düzenli bir kullanıcı olmak daha iyidir; kök haline gelmeniz gerekirse dikkatli olun, çünkü bilgi işlem dünyasının yıkıcı güçlerinin hepsi artık parmaklarınızın ucunda.

işlemlere komut sinyalleri de gönderilir. Bunları dikkatli bir şekilde kullandığınızdan emin olun; pencere yöneticinizi yanlışlıkla öldürürseniz, önünüzde oturduğunuz bilgisayarı kullanmak oldukça zor olabilir. Son olarak, sisteminizdeki yükü hızlı bir şekilde anlamak için kullanabileceğiniz birçok farklı CPU ölçüm birimi vardır; Örneğin, her zaman **MenuMeters'i** (Raging Menace Software'den) Macintosh araç çubuklarımızda çalışır durumda tutuyoruz, böylece her an ne kadar CPU kullanıldığını görebiliyoruz. Genel olarak, ne hakkında daha fazla bilgi

devam edin, daha iyi.

5.7 Özet

UNIX işlem oluşturma ile ilgili bazı API'leri kullanıma sunduk: `fork()`, `exec()`, and `wait()`. Ancak, sadece yüzeyi gözden geçirdik. Daha fazla ayrıntı için, Stevens ve Rago [SR05] bölümlerini, özellikle Süreç Kontrolü, Süreç İlişkileri ve Sinyallerle ilgili bölümleri okuyun; bilgelikten çıkarmanız gereken çok şey var.

UNIX Process API'ye olan tutkumuz güçlü olmakla birlikte, bu tür pozitifliğin tek tip olmadığını da unutmamalıyız. Örneğin, Microsoft, Boston Üniversitesi ve İsviçre'deki ETH'den sistem araştırmacıları tarafından hazırlanan yakın tarihli bir makalede `fork()` ile ilgili bazı sorunlar ayrıntılı olarak ele alınmaktadır ve **`spawn()`** [B+19] gibi daha basit proses oluşturma API'lerinin savunucuları bulunmaktadır. Bu farklı bakış noktasını anlamak için bu noktayı ve ilgili çalışmayı okuyun. Bu kitaba güvenmek genellikle iyi olsa da, yazarların fikirlerinin de olduğunu unutmayın; bu görüşler (her zaman) düşündüğün kadar yaygın olmayabilir.

Not: Anahtar işlem API koşulları

- Her sürecin bir adı vardır; çoğu sistemde bu ad, **Süreç Kimliği (PID)** olarak bilinen bir numaradır.
- **fork()** sistem çağrısı, UNIX sistemlerinde yeni bir pro-cess oluşturmak için kullanılır. İçerik oluşturucuya **üst öğe** (**parent**) adı verilir; yeni oluşturulan işleme **alt öğe** (**child**) adı verilir. Bazen gerçek hayatta olduğu gibi [J16], alt öğe süreci üst öğenin neredeyse aynı bir kopyasıdır.
- **wait()** sistem çağrısı, bir ebeveynin, alt öğenin yürütmeyi tamamlamasını beklemesini sağlar.
- **exec()** sistem çağrı ailesi, bir çocuğun ana öğeye benzerliğinden kurtulmasını ve tamamen yeni bir program yürütmesini sağlar.
- Bir UNIX **kabuğu (shell)** genellikle kullanıcı komutlarını başlatmak için `fork()`, `wait()`, and `exec()` komutlarını kullanır; fork ve exec ayrımı, çalıştırılan programlarla ilgili herhangi bir değişiklik yapmadan **giriş/çıkış yeniden yönlendirme (input/output redirection), veriyolları (pipes)** ve diğer havalı özellikler gibi özellikleri etkinleştirir.
- Süreç kontrolü, işlerin durdurulmasına, devam etmesine ve hatta sonlandırılmasına neden olabilecek **sinyal (signals)** biçiminde mevcuttur.
- Belirli bir kişi tarafından kontrol edilebilecek işlemler **kullanıcı** (**user**) kavramıyla kaplanmıştır; işletim sistemi sistemde birden fazla kullanıcıya izin verir ve kullanıcıların yalnızca kendi süreçlerini kontrol edebilmelerini sağlar.
- Bir **yetkili kullanıcı (superuser)** tüm süreçleri kontrol edebilir (ve gerçekten de birçok başka şeyi yapabilir); bu rol nadiren ve güvenlik nedenleriyle dikkatli bir şekilde üstlenilmelidir.

Referanslar

[B+19] "A fork() in the road" by Andrew Baumann, Jonathan Appavoo, Orran Krieger, Timothy Roscoe. HotOS '19, Bertinoro, Italy. *A fun paper full of `fork()`ing rage. Read it to get an opposing viewpoint on the* UNIX *process API. Presented at the always lively HotOS workshop, where systems researchers go to present extreme opinions in the hopes of pushing the community in new directions.*

[C63] "A Multiprocessor System Design" by Melvin E. Conway. AFIPS '63 Fall Joint Computer Conference, New York, USA 1963. *An early paper on how to design multiprocessing systems; may be the first place the term `fork()` was used in the discussion of spawning new processes.*

[DV66] "Programming Semantics for Multiprogrammed Computations" by Jack B. Dennis and Earl C. Van Horn. Communications of the ACM, Volume 9, Number 3, March 1966. *A classic paper that outlines the basics of multiprogrammed computer systems. Undoubtedly had great influence on Project MAC, Multics, and eventually* UNIX.

[J16] "They could be twins!" by Phoebe Jackson-Edwards. The Daily Mail. March 1, 2016.. *This hard-hitting piece of journalism shows a bunch of weirdly similar child/parent photos and is frankly kind of mesmerizing. Go ahead, waste two minutes of your life and check it out. But don't forget to come back here! This, in a microcosm, is the danger of surfing the web.*

[L83] "Hints for Computer Systems Design" by Butler Lampson. ACM Operating Systems Review, Volume 15:5, October 1983. *Lampson's famous hints on how to design computer systems. You should read it at some point in your life, and probably at many points in your life.*

[QI15] "With Great Power Comes Great Responsibility" by The Quote Investigator. Available: `https://quoteinvestigator.com/2015/07/23/great-power`. *The quote investigator concludes that the earliest mention of this concept is 1793, in a collection of decrees made at the French National Convention. The specific quote: "Ils doivent envisager qu'une grande responsabilite´ est la suite inse´parable d'un grand pouvoir", which roughly translates to "They must consider that great responsibility follows inseparably from great power." Only in 1962 did the following words appear in Spider-Man: "...with great power there must also come–great responsibility!" So it looks like the French Revolution gets credit for this one, not Stan Lee. Sorry, Stan.*

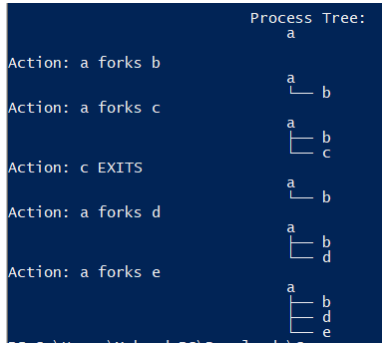
[SR05] "Advanced Programming in the UNIX Environment" by W. Richard Stevens, Stephen A. Rago. Addison-Wesley, 2005. *All nuances and subtleties of using* UNIX *APIs are found herein. Buy this book! Read it! And most importantly,* ***live it.***

Ödev (Simülasyon)

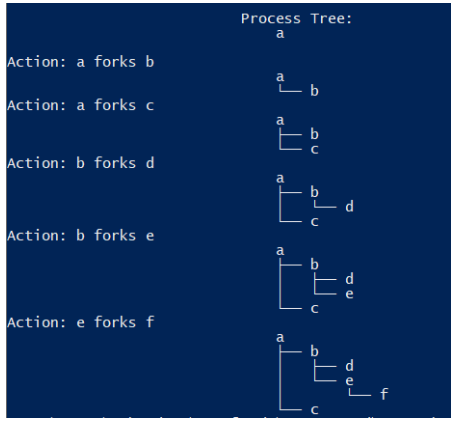
Bu simülasyon ev ödevi, süreçlerin tek bir "aile" ağacında nasıl ilişkili olduğunu gösteren basit bir süreç oluşturma simülatörü olan fork.py'ye odaklanır. Simülatörün nasıl çalıştırılacağını öğrenmek için ilgili BENIOKU DOSYASINI okuyun.

Sorular

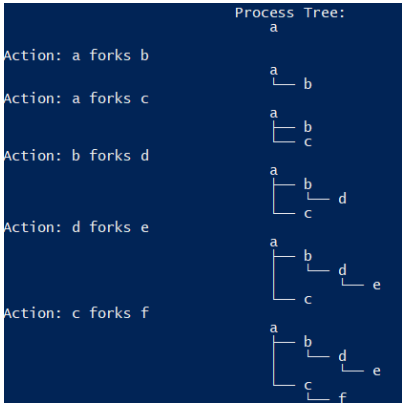
1. Çalıştırın./fork.py-s 10 ve hangi eylemlerin alındığını görün. Süreç ağacının her adımda nasıl görüneceğini tahmin edebilir misiniz? Yanıtlarınızı kontrol etmek için -c bayrağını kullanın. Farklı rastgele tohumlar (-s) deneyin veya daha fazla eylem (-a) ekleyerek bunun nasıl çalıştığına bakın?



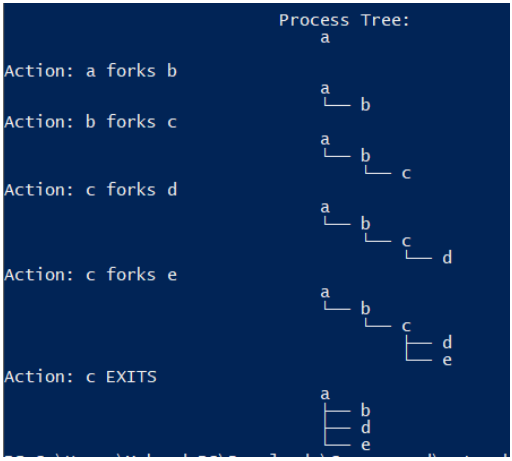
2. Simülatörün size sağladığı kontrollerden biri, -f _ bayrağı tarafından kontrol edilen `fork` yüzdesidir. Ne kadar yüksek olursa, bir sonraki eylem fork olur; ne kadar düşük olursa, eylem bir çıkış olur. Simülatörü çok sayıda işlem (örn. -A 100) ile çalıştırın ve `fork` yüzdesini 0.1'den 0.9'e değiştirin. Sizce ortaya çıkan son süreç ağaçları yaş yüzdesi değiştikçe nasıl görünür? -c ile yanıtınızı kontrol edin.



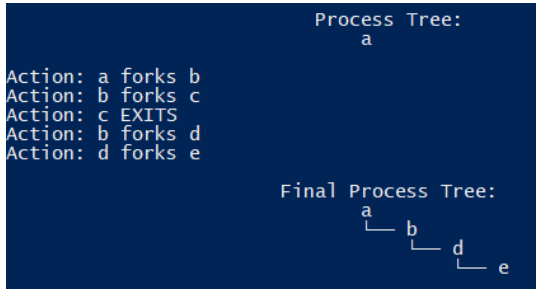
3. Şimdi -t işaretini kullanarak çıkışı değiştirin (örn. Run./fork.py-t). Süreç ağaçları seti göz önünde bulundurulduğunda hangi eylemlerin alındığını söyleyebilir misiniz?



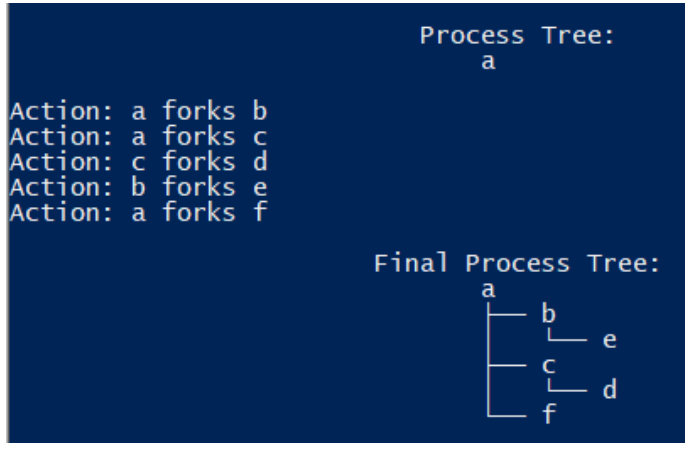
4. Dikkat edilmesi gereken ilginç bir nokta, bir alt öğe çıkınca ne olur; süreç ağacındaki alt öğelerine ne olur? Bunu incelemek için, özel bir örnek oluşturalım: `./fork.py -A a+b,b+c,c+d,c+e,c-`. Bu örnekte, (a) oluşturma (b) süreci vardır ve bu süreç (c) oluşturur, ardından (d) ve (e) oluşturur. Ancak, (c)'den çıkılır. Çıkış sonrasında süreç ağacının nasıl olması gerektiğini düşünüyorsunuz? (-R) bayrağını kullanırsanız ne olur? Daha fazla bağlam eklemek için tek başınıza artık süreçlere ne olduğu hakkında daha fazla bilgi edinin.



5. O Keşfedilecek son bayraklardan biri, ara adımları atlayıp sadece son işlem ağacını doldurmayı isteyen -F bayrağıdır. `./fork.py -F`'yi çalıştır ve oluşturulan eylem dizilerine bakarak son ağacı not edip edemediğinizi görün. Bunu birkaç kez denemek için farklı rastgele tohumlar kullanın.



6. Son olarak, hem -t hem de -F'yi birlikte kullanın. Bu, son işlem ağacını gösterir ancak ardından gerçekleştirilen eylemleri doldurmanızı ister. Ağaca bakarak, yapılan işlemlerin tam olarak ne olduğunu belirleyebilir misiniz? Hangi durumlarda söyleyebilirsiniz? Hangisinde bunu söyleyemiyorsun? Bu soruyu araştırmak için farklı rastgele tohumlar deneyin.



Not : Kodlama Ödevleri

Kod yazma ev işleri, bazı temel işletim sistemi API'leri ile ilgili deneyim elde etmek için gerçek bir makinede kod yazmak üzere yazdığınız küçük alıştırmalar olarak kullanılır. Sonuçta (muhtemelen) bir bilgisayar bilim insanı olduğunuz için kod yazmayı sevmeniz gerekir, değil mi? Bunu yapmazsanız HER zaman CS teorisi vardır, ancak bu oldukça zordur. Tabii ki gerçek bir uzman olmak için makinede bilgisayar korsanlığı yapmak için biraz zaman harcamanız gerekir; gerçekten de, bir kod yazmak ve nasıl çalıştığını görmek için her türlü bahaneyi bulabilirsiniz. Zaman ayırın ve bildiğiniz bilge usta olun.

Ödev (Kod)

Bu ev ödevinde, az önce okuduğunuz süreç yönetimi API'lerini biraz iyi tanımanız gerekir. Endişelenmeyin; seslerinden daha da eğlenceli! Kod yazmak için olabildiğince fazla zaman bulursanız genel olarak çok daha iyi olacaksınız, öyleyse şimdi başlamaya ne dersiniz?

Sorular

1. `fork()` olarak adlandırdığı bir program yazın. `fork()` aramadan önce, ana işlemin bir değişkene (örn. X) erişmesini sağlayın ve değerini bir şeye (örn., 100) ayarlayın. Alt öğe sürecindeki değişken nedir? Hem alt hem de üst öğe x değerini değiştirdiğinde değişkene ne olur?

   Aşağıda, fork() çağrıldıktan sonra hem üst öğe hem de alt öğe süreci tarafından erişildiğinde değişkenin davranışını gösteren bir örnek program bulunmaktadır:

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    int x = 100; // initialize x to 100

    printf("x: %d\n", x); // print the initial value of x

    pid_t pid = fork(); // create a child process

    if (pid == 0) // this block runs in the child process
    {
        x = 50; // change the value of x in the child process

        printf("[child] x: %d\n", x); // print the value of x in the child process
    }
    else // this block runs in the parent process
    {
        x = 25; // change the value of x in the parent process

        printf("[parent] x: %d\n", x); // print the value of x in the parent process

        wait(NULL); // wait for the child process to finish
    }

    return 0;
}
```

fork()'ı aradıktan sonra, alt öğe işlemi, x değişkeninin değeri de dahil olmak üzere ana işlemin belleğinin bir kopyasına sahiptir. Bu durumda, x'in başlangıç değeri 100'dir. Bu değer hem ana hem de alt işlemlerle aynıdır.

Ancak, üst işlemde x değeri değiştirildiğinde, alt işlemin kendi belleği olduğundan alt işlemde değişiklik görünmez. Benzer şekilde, alt işlemde x değeri değiştirildiğinde, değişiklik üst işlemde görünmez.

Program çalışırken aşağıdaki çıktıyı yazdırır:

```c
x: 100
[child] x: 50
[parent] x: 25
```

Bu, her işlemde değiştirildikten sonra üst ve alt işlemlerde x değerinin farklı olduğunu gösterir.

2. Bir dosya (`open()` sistem çağrısı ile) açan bir program yazın ve ardından yeni bir işlem oluşturmak için `fork()` arar. Hem alt hem de üst öğe `open()` tarafından döndürülen dosya tanımlayıcıya erişebilir mi? Aynı anda dosyaya yazdıklarında ne olur, yani aynı anda?

Evet, hem alt öğe hem de üst öğe süreci fork() aradıktan sonra open() tarafından döndürülen dosya tanımlayıcıya erişebilir. Ancak, aynı anda dosyaya yazmaya çalışırlarsa, yazdıklarının yürütüleceği sipariş hakkında herhangi bir garanti yoktur. Bu durum beklenmeyen davranışlara ve olası veri bozulmasına neden olabilir, çünkü yazılar amaçlanmayan bir şekilde birbirine geçmiş olabilir.

Aşağıda bu davranışı gösteren örnek bir program verilmiştir:

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <fcntl.h>

int main()
{
    int fd = open("file.txt", O_WRONLY | O_CREAT, 0644); // open a file for writing

    pid_t pid = fork(); // create a child process

    if (pid == 0) // this block runs in the child process
    {
        write(fd, "child\n", 6); // write "child" to the file
    }
    else // this block runs in the parent process
    {
        write(fd, "parent\n", 7); // write "parent" to the file

        wait(NULL); // wait for the child process to finish
    }

    close(fd); // close the file

    return 0;
}
```

Bu program çalışırken aynı anda dosyaya "alt" ve "üst" yazacaktır. Yazıların birbirine nasıl geçtiğine bağlı olarak, dosyanın içeriği "alt\nüst\n", "üst\nalt\n" veya başka bir kombinasyon olabilir.

Bu sorunu önlemek için, dosyaya belirli bir anda yalnızca bir işlemin yazılmasını sağlamak üzere mutesler veya semafolar gibi senkronizasyon mekanizmalarının kullanılması önerilir. Alternatif olarak işlemler, yazmadan önce dosya ofsetini dosyanın sonuna taşımak için lseek () özelliğini kullanabilir, böylece yazmaları birbirlerinin verilerinin üzerine yazmaz.

3. `fork()` kullanarak başka bir program yazın. Çocuk işlemi "merhaba" olarak yazdırmalıdır; ana işlem "güle güle" olarak yazdırmalıdır. Alt işlemin her zaman ilk olarak yazdırılmasını sağlamaya çalışmalısınız; bunu, ana işlemdeki **wait()** mesajını aramadan yapabilir misiniz?

fork() kullanarak başka bir program yazın. Alt öğe işlemi "merhaba" olarak yazdırmalıdır; ana işlem "güle güle" olarak yazdırmalıdır. Alt işlemin her zaman ilk olarak yazdırılmasını sağlamaya çalışmalısınız; bunu, ana işlemdeki wait() mesajını aramadan yapabilir misiniz?

Evet, ebeveyn işlemi ana işlemdeki beklemeden () "güle güle" yazmadan önce çocuk işleminin her zaman "merhaba" olarak yazdırmasını sağlamak mümkündür. Bu, ana işlemdeki sleep() işlevi çağrılarak, yürütülmesini geciktirilerek alt öğe işleminin önce mesajını yazdırmasına olanak tanınabilir.

Aşağıda bu davranışı gösteren örnek bir program verilmiştir:

```
   c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    pid_t pid = fork(); // create a child process

    if (pid == 0) // this block runs in the child process
    {
        printf("hello\n"); // print "hello"
    }
    else // this block runs in the parent process
    {
        sleep(1); // delay the execution of the parent process

        printf("goodbye\n"); // print "goodbye"
    }

    return 0;
}
```

Bu program çalışırken önce "merhaba", ardından "güle güle" yazacaktır. sleep () işlevi, üst işlemin belirli bir süre duraklamasına neden olarak, alt işlemin ana işlem devam etmeden önce çalışmasını ve mesajını yazdırmasını sağlar.

Bu yaklaşımın güvenilir olmadığını belirtmek önemlidir çünkü sistem yükü ve zamanlama gibi faktörlere bağlı olarak değişebilen süreçlerin zamanlamasına dayanır. Alt öğe işleminin her zaman ilk olarak yazdırılmasını sağlamanın daha güvenilir bir yolu, mütesler veya semafores gibi senkronizasyon mekanizmalarını kullanmaktır.

4. Çatalı () arayan bir program yazın ve programı/ambar/ls çalıştırmak için bir miktar exec () araması. (Linux üzerinde) yürüt (), yürüt (), yürüt (), yürüt (), yürüt (), yürüt (), yürüt (), yürüt (), yürüt (), ve yürütür (). Sizce neden aynı temel aramanın bu kadar çok çeşidi var?

   Programı/bin/ls'yi çalıştırmak için fork() ve ardından exec() adındaki basit bir program:

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
  int pid = fork();
  if (pid == 0) {
    // This is the child process.
    // Use exec() to run /bin/ls.
    execl("/bin/ls", "/bin/ls", NULL);
  } else {
    // This is the parent process.
    // Wait for the child process to finish.
    int status;
    wait(&status);
    printf("Child process finished with status %d\n", status);
  }

  return 0;
}
```

   Duruma bağlı olarak farklı şekillerde kullanılabileceği için exec() çağrının birçok çeşidi vardır. Örneğin, bazı varyantlar komutu ve bağımsız değişkenlerini tek bir dize olarak değil, ayrı dizeler olarak belirtmenize olanak tanır; bu, bazı durumlarda daha kullanışlı olabilir. Ayrıca, bazı varyantlar komut için farklı arama yolları belirlemenizi veya yeni işlem için farklı ortam değişkenleri belirlemenizi sağlar. Farklı exec() sürümleri, kendi programınızdan diğer programları çalıştırmak için farklı seçenekler ve esneklik sağlar.

5. Şimdi, alt öğenin işleminin üst öğeden bitmesini beklemek için `wait()` özelliğini kullanan bir program yazın. `wait()` ne işe yarıyor? Alt öğede `wait()` seçeneğini kullanırsanız ne olur?

Aşağıda, alt öğenin işlemini bitirmesini beklemek için wait() özelliğini kullanan önceki programın değiştirilmiş bir sürümü bulunmaktadır:

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
  int pid = fork();
  if (pid == 0) {
    // This is the child process.
    // Use exec() to run /bin/ls.
    execl("/bin/ls", "/bin/ls", NULL);
  } else {
    // This is the parent process.
    // Wait for the child process to finish.
    int status;
    wait(&status);
    printf("Child process finished with status %d\n", status);
  }

  return 0;
}
```

wait() fonksiyonu, alt öğe işleminin çıkış durumunu geri döndürür. Alt öğe işlemi normal şekilde sonlandırılırsa çıkış durumu, alt öğe işleminin exit() fonksiyonuna geçtiği değer olur. Alt öğe işlemi bir sinyalle sonlandırılırsa çıkış durumu, alt işlemi sonlandıran sinyali temsil eden negatif bir sayı olur.

Alt süreçte bekle() kullanıyorsanız, bu işlem tüm kardeş süreçlerinin (yani aynı ana işlem tarafından oluşturulan işlemler) sona ermesini bekler. Bu genellikle istediğiniz şey değildir, çünkü çocuk süreci zaten execl() çağrının bitmesini bekliyor olacaktır, bu yüzden wait() Çocuk sürecindeki arama gereksiz ek ek yük ekler ve ek faydalar sağlamaz.

6. Bu sefer `wait()` yerine `waitpid()` kullanarak önceki programda yapılan küçük değişiklikleri yazın. `waitpid()`ne zaman faydalı olur?

   Aşağıda, alt öğe işleminin bitmesini beklemek için bekle() yerine waitpid () kullanan, önceki programın değiştirilmiş bir sürümü bulunmaktadır:

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
  int pid = fork();
  if (pid == 0) {
    // This is the child process.
    // Use exec() to run /bin/ls.
    execl("/bin/ls", "/bin/ls", NULL);
  } else {
    // This is the parent process.
    // Wait for the child process to finish.
    int status;
    waitpid(pid, &status, 0);
    printf("Child process %d finished with status %d\n", pid, status);
  }

  return 0;
}
```

   waitpid() işlevi wait() işlevine benzer, ancak beklemek istediğiniz alt öğe işleminin proses kimliğini belirlemenize olanak sağlar. Bu, programınız birden fazla alt öğe işlemi oluşturduğunda ve herhangi bir alt öğe işleminin bitmesini beklemek yerine belirli bir alt öğe işleminin bitmesini beklemek istediğinizde yararlı olabilir. Buna ek olarak, waitpid(), waitpid() 'in nasıl davranacağını kontrol eden ek seçenekler belirtmenize olanak tanır. Örneğin, alt öğe işlemi henüz bitmezse hemen geri dönüp dönmemesi veya alt öğe işleminin bitmesini bekleyip beklememesi gibi.

7. Alt öğe süreci oluşturan bir program yazın ve alt öğe içinde standart çıkışı kapatır (STDOUT FILENO). Alt öğe, tanımlayıcı kapatıldıktan sonra çıktı yazdırmak için `printf()`'i ararsa ne olur?

   Aşağıda alt öğe süreci oluşturan ve alt öğe sürecinde standart çıkışı kapatan basit bir program bulunmaktadır:

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

int main() {
  int pid = fork();
  if (pid == 0) {
    // This is the child process.
    // Close standard output.
    close(STDOUT_FILENO);

    // Attempt to print some output.
    printf("Hello from the child process\n");
  } else {
    // This is the parent process.
    // Wait for the child process to finish.
    int status;
    waitpid(pid, &status, 0);
    printf("Child process %d finished with status %d\n", pid, status);
  }

  return 0;
}
```

Alt öğe işlemi, standart çıktı tanımlayıcısını kapattıktan sonra bir miktar çıktı yazdırmayı denerse alt öğe işlemi terminale işaret eden bir açık dosya tanımlayıcısına sahip olmadığından çıktı ekranda görüntülenmez. Bu, alt işlemdeki çıktıyı bastırmak istediğinizde veya çıktıyı bir dosyaya veya başka bir aygıta yeniden yönlendirmek istediğinizde yararlı olabilir. Alt işlemin hata mesajlarını standart hata akışına (genellikle terminalde de görüntülenir) yazdırmaya devam edebileceğini unutmayın; bu nedenle, alt işlemdeki tüm çıktıları tamamen bastırabilmek için standart hata akışını yeniden yönlendirmeniz veya bastırmanız gerekebilir.

8. İki alt öğe oluşturan bir program yazın ve `pipe()` sistem çağrıyla bir alt öğenin standart çıktısını diğerinin standart girişine bağlar.

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(int argc, char **argv)
{
    // Create a pipe
    int pipefd[2];
    if (pipe(pipefd) == -1)
    {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    // Create the first child process
    int child1 = fork();
    if (child1 == -1)
    {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    else if (child1 == 0)
    {
        // In the first child process, close the read end of the pipe
        close(pipefd[0]);

        // Write some data to the pipe
        write(pipefd[1], "hello", 5);

        // Close the write end of the pipe
        close(pipefd[1]);
    }
    else
    {
        // Create the second child process
        int child2 = fork();
        if (child2 == -1)
        {
            perror("fork");
            exit(EXIT_FAILURE);
        }
        else if (child2 == 0)
        {
            // In the second child process, close the write end of the pipe
            close(pipefd[1]);

            // Read some data from the pipe
            char buf[5];
            read(pipefd[0], buf, 5);
            printf("buf = %s", buf);

            // Close the read end of the pipe
            close(pipefd[0]);
        }
        else
        {
            // In the parent process, close both ends of the pipe
            close(pipefd[0]);
            close(pipefd[1]);

            // Wait for the first child process to exit
            waitpid(child1, NULL, 0);

            // Wait for the second child process to exit
            waitpid(child2, NULL, 0);
        }
    }
}
```